Claremont Colleges
Scholarship @ Claremont

All HMC Faculty Publications and Research | HMC Faculty Scholarship

11-1-1983

Specification of Synchronizing Processes

Krithivasan Ramamritham
University of Massachusetts - Amherst

Robert M. Keller
Harvey Mudd College

Recommended Citation

Ramamritham, K., and R.M. Keller. "Specification of synchronizing processes." IEEE Transactions on Software Engineering Vol. SE-9, Issue 6 (November 1983): 722-733. DOI: 10.1109/TSE.1983.235435

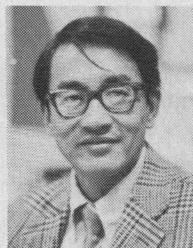
This Article is brought to you for free and open access by the HMC Faculty Scholarship at Scholarship @ Claremont. It has been accepted for inclusion in All HMC Faculty Publications and Research by an authorized administrator of Scholarship @ Claremont. For more information, please contact scholarship@cuc.claremont.edu.

[9] D. I. Good *et al.*, "Principles of proving concurrent programs in GYPSY," Univ. Texas, Austin, Tech. Rep. ICSCA-CMP-15, Jan. 1979.
[10] I. Greif, "A language for formal problem specification," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 931-935, Dec. 1979.
[11] B. Hailpern and S. Owicki, "Verifying network protocols using temporal logic," in *Proc. Trends and Appl. 1980: Comput. Network Protocols*, IEEE Comput. Soc., May 1980.
[12] C. Hewitt and H. J. Baker, "Laws for communicating parallel processes," *IFIP*, pp. 987-992, 1977.
[13] C. A. R. Hoare, "A model for communicating sequential processes," Comput. Lab., Oxford Univ., Dec. 1978.
[14] L. Lamport, "Time clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558-565, July 1978.
[15] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 417-526, July 1981.
[16] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Programming Lang. Syst.*, vol. 4, pp. 455-495, July 1982.
[17] J. L. Peterson, "Petri nets," *ACM Comput. Surveys*, vol. 9, pp. 223-253, Sept. 1977.
[18] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symp. Foundation of Comput. Sci.*, Province, Nov. 1977, pp. 46-57.
[19] L. Robinson and D. Roubine, "SPECIAL: A SPECIfication and Assertion Language," Stanford Res. Inst., Tech. Rep. CSL-46, 1977.
[20] R. L. Schwartz, and P. M. Melliar-Smith, "Temporal logic specification of distributed systems," in *Proc. 2nd Int. Conf. Distributed Comput. Syst.*, Paris, France, Apr. 1981, pp. 446-454.
[21] N. V. Stenning, "A data transfer protocol," *Comput. Networks*, vol. 1, pp. 99-110, Sept. 1976.
[22] C. Sunshine, "Formal methods for communication protocol specification and verification," Rand Corp., Working Draft WD-335-ARPA/NBS, Sept. 1979.
[23] R. T. Yeh and P. Zave, "Specifying software requirements," *Proc. IEEE*, Oct. 1980.
[24] P. Zave and R. T. Yeh, "Executable requirements for enbedded systems," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, 1981.
[25] C. C. Zhoa and C. A. R. Hoare, "Partial correctness of communicating sequential processes," in *Proc. 2nd Int. Conf. Distributed Comput. Syst.*, Paris, France, Apr. 1981, pp. 1-12.

Bo-Shoe Chen (S'79-M'82) received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, in 1975, and the M.S. and the Ph.D. degrees in computer science from the University of Maryland, College Park, in 1980 and 1982, respectively.

He has worked on the design, development, modeling, and performance measurements of Bell system packet-switched networks since he joined Bell Laboratories, Naperville, IL in 1982. His current interests include software tools, distributed computing, and integrated service of digital networks. He was an instructor in the University College, University of Maryland, Hyttsville, from 1980 to 1981; and worked on Chinese input/output processors for a database system in Taiwan Automation Corporation before he came to the U.S. in 1978.

Dr. Chen is a member of the IEEE Computer Society.



Raymond T. Yeh (S'64-M'72-SM'78-F'83) is currently a Professor of Computer Science at the University of Maryland, College Park. He is also sharing the CDC Endowed Chair of Distinguished Professor in Computer Science at the University of Minnesota this year. He has been Department Chairman both at the University of Texas at Austin, and at Maryland. His current research interests include programming environment, distributed software, and software maintenance methodology. He was founding Editor-in-Chief of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Specification of Synchronizing Processes

KRITHIVASAN RAMAMRITHAM AND ROBERT M. KELLER

***Abstract*—The formalism of temporal logic has been suggested to be an appropriate tool for expressing the semantics of concurrent programs. This paper is concerned with the application of temporal logic to the specification of factors affecting the synchronization of concurrent processes. Towards this end, we first introduce a model for synchronization and axiomatize its behavior. SYSL, a very high-level language for specifying synchronization properties, is then described. It is designed using the primitives of temporal logic and features constructs to express properties that affect synchronization in a fairly natural and modular fashion. Since the statements in the language have intuitive interpretations, specifications are humanly readable. In addition, since they possess appropriate formal semantics, unambiguous specifications result.**

***Index Terms*—Abstract model, concurrent processing, specification language, synchronization, temporal logic.**

Manuscript received June 15, 1981; revised March 15, 1983. This work was supported by the National Science Foundation under Grants MCS-77-09369 and MCS-82-02586. A preliminary version of this paper appeared as [27].

K. Ramamritham is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.
R. M. Keller is with the Department of Computer Science, University of Utah, Salt Lake City, UT 84112.

I. INTRODUCTION

PROCESSES executing in parallel interact with each other either by sending messages or by modifying states of shared resources. When *message passing* is the means of interaction, one process sends a message to another and synchronization of the receiving process occurs through the message [15]. On the other hand, processes that communicate by *sharing resources* do so by executing specific operations which change

0098-5589/83/1100-0722$01.00 © 1983 IEEE

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

the state of the resource. Execution of these operations is *synchronized* by some appropriate mechanism, e.g., monitors [14]. As a result, the processes that share the resource are also synchronized. In this paper we address issues related to the shared resource paradigm of interaction.

To maintain the integrity of a shared resource, an answer to the question, "*who* is to access the resource, *when*, and *how*?," is essential. A protection mechanism is responsible for *who* accesses the resource and a typing mechanism for *how* the resource is accessed. On the other hand, the synchronizer is responsible for *when* the access actually takes place. This paper concerns only the synchronization of accesses to a shared resource. In particular, we are interested in

1) modeling the behavior of a synchronizer of concurrent processes and
2) designing a language to specify the synchronization of concurrent processes accessing a shared resource.

The requisite formalism is provided by *temporal logic*, its chief advantage being that it facilitates a unified approach to specification and verification of both safety and liveness properties of software systems [18], [24].

Our view of synchronization is based on the assumption of one synchronizer per shared resource. All accesses to a shared resource are controlled by the synchronizer for that resource. In this model an access operation goes through four phases. They are the *request* phase, the *service* phase, the *active* phase, and the *termination* phase. The model takes into accout the temporal ordering of the phases of operations when users make concurrent requests. In Section II, the behavior of a synchronizer that conforms to such a model is axiomatized by means of temporal logic assertions.

Section III introduces SYnchronizer Specification Language (SYSL) which includes constructs designed to express various aspects of synchronization control, such as constraints governing access to shared resources, priority of various types of access, mutual exclusion of access, invariance of the resource state, absence of starvation, and other relevant properties. Constructs in SYSL have intuitive interpretations and due to their temporal logic basis possess unambiguous semantics. Thus, our approach to specification of synchronization is at a level reasonably close to a human conceptual model. One of the motivations behind the design of SYSL is the automatic synthesis of synchronization code [28]. Thus, language features in SYSL are motivated by the need to provide programmers an easy to use specification language and to facilitate the synthesis of synchronization code. Evaluation of the language features precedes concluding remarks on our approach to the specification of synchronization.

II. The Semantics of Synchronizers

This section formalizes the notion of a *synchronizer*. We introduce an operational model for a synchronizer and define the terms associated with the domain of synchronization. This sets the stage for the development of the specification language in the next section.

A. Temporal Logic

Pnueli first applied temporal logic for reasoning about safety and liveness properties of concurrent programs [24], [25]. Following along those lines, concurrency is modeled by a nondeterministic interleaving of computations of individual processes. Each computation changes the system *state* which consists of values assigned to program variables and the instruction pointer of each of the processes. Using temporal logic operators, one can specify and reason about the properties of the sequence of states that results from the execution of the concurrent processes.

Since temporal logic is an extension of predicate calculus, a temporal logic statement can involve the usual logical operators $\vee$ (or), $\wedge$ (and), $\neg$ (not), and $\Rightarrow$ (implication) besides the temporal operators $\Box$, $\Diamond$, and UNTIL. The operator $\Box$ is pronounced "always." $\Box P$ states that P is true now and will remain true throughout the future. The operator $\Diamond$ is pronounced "eventually" and is the dual of $\Box$ in that

$$\Diamond P \text{ IFF } \neg \Box \neg P.$$

Thus, $\Diamond P$ if P is true now or will be true sometime in the future. A requirement such as "every request *will* be serviced" can be specified as

$$\Box\{\text{"request for service exists"} \Rightarrow \Diamond \text{ "request serviced"}\}.$$

The operator UNTIL has the following interpretation:

(P UNTIL Q) IFF P will be true as long as Q is false.

(The truth value of P once Q becomes true is not indicated by UNTIL.) The UNTIL operator is typically used for expressing temporal orderings. For example, the fact that a service can not be provided until there is a request for that service can be stated as

$\neg$("request serviced") UNTIL ("request for service exists")

The semanitcs of $\Box$ and $\Diamond$ are identical to those of the corresponding linear time logic operators of [18] whereas UNTIL is related to Lamport's binary $\Box$ operator (read AS LONG AS) in the following manner:

$$(P \text{ UNTIL } Q) \equiv (\neg Q \Box P)$$

Also, our definition of UNTIL does not assert that eventually Q becomes true and hence differs from the definition of UNTIL given in [25].

Certain operators are derived from these primitives, and are introduced to enhance the readability of specifications in SYSL. They are

P ONLYAFTER Q	($\neg$P UNTIL Q), P *can* become true only after Q does.
P AFTER[1] Q	($\neg$P UNTIL Q) $\wedge$ $\Diamond$P i.e., P *will* become true after Q.

where P and Q are arbitrary assertions.

B. The Concept of a Synchronizer

By a *synchronizer*, we mean a *sequential* process which guarantees disciplined access to a *shared resource*. Each distinct *type of access* on the resource is called an *operation class*. All instances of a particular type are said to be *operations* in that operation class. Any access to the resource is through the

[1] In fact, given our definition of UNTIL, P AFTER Q asserts that P will become true the same time as or after Q.

Authorized licensed use limited to: Claremont Colleges. Downloaded on June 11, 2009 at 07:52 from IEEE Xplore. Restrictions apply.

execution of one of the operations. Furthermore, each of the operations can execute only when the synchronizer permits it to do so.

Constraints essential for maintaining the integrity of a resource are built into a synchronizer for it. Thus, it is useful to perceive the synchronizer as a process guarding a shared resource from improper use by the concurrent processes. In short, a shared resource comprises of the following:

- the *resource* that is shared,
- the *operations* on the resource, and
- the *synchronizer* of the operations.

Hence a shared resource can be considered to be an abstract data type [11] with additional synchronization restrictions. Execution of an operation goes through four distinct phases in sequence: Request: Service, Active, and Termination. The *request phase* for an operation begins after a synchronizer recognizes that a user program requests execution of that operation. The request phase ends when the synchronizer's internal data structures reflect the fact that a request is waiting for service. The time at which the synchronizer permits execution of a requested operation depends on the state of the shared resource, priority associated with the request, invariant properties of the resource, etc. These determine the *necessary conditions* for executing an operation. The *service phase* begins when and if the necessary conditions hold and the synchronizer decides to permit the execution of the operation. Thus, by requiring that the necessary conditions must hold when the operation is serviced, the synchronizer guarantees that the specified properties are maintained. At the end of the service phase, the synchronizer's internal data structures reflect the fact that permission has been granted for the execution of the operation. Thus the term "service" is equivalent to "granting of permission." The *active phase* begins after the service phase ends. It is in this phase that the resource access defined by the operation takes place. The active phase ends when access is complete. The *termination phase* begins after the active phase ends. At the end of the termination phase the synchronizer's internal data structures reflect the fact that the operation has completed execution.

The synchronizer has been described as a sequential process. This implies that there is a single locus of control within the synchronizer. From an operational viewpoint, all actions of the synchronizer are *serialized.* Thus, since one of the main actions of the synchronizer is to service requests, only one request can be serviced at a time. However, at any given time, two or more operations may have satisfied their necessary conditions, in which case, the synchronizer chooses one among them to be serviced next. This choice may be made to obey a fairness specification.

C. A Formal Model for a Synchronizer

To precisely define the model, we introduce some notation.

p|op

will be used to refer to phase p of operation op. If process pr executes phase p of operation op,

at (p|op) IFF control of pr is at the beginning of p.

in (p|op) IFF control of pr is within p.

after (p|op) IFF control of pr is at the end of p.

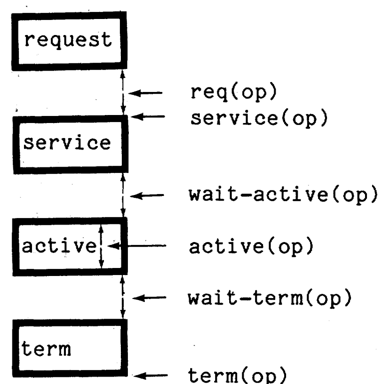


Fig. 1. Sequence of phases of an operation.

These three predicates are mutually exclusive and become true in the above order. Now we examine the temporal ordering of phases of operations. The four phases associated with any operation "op" are totally ordered in time as follows:

at (service|op) ONLYAFTER after (request|op)

at (active|op) ONLYAFTER after (service|op)

at (term|op) ONLYAFTER after (active|op).

The use of ONLYAFTER (instead of IFF) in the above statements reflects the possibility of delays between the execution of two consecutive phases. The above statements, in addition to the fact that

after (p|op) ONLYAFTER at (p|op)

for all phases p of operation op, define the sequential ordering of the phases of any operation op.

To precisely specify the state of each operation we introduce the following predicates:

req (op)	is true when operation op is waiting to be serviced.
service (op)	is true when permission is granted for executing a.
wait-active (op)	becomes true after op is serviced and remains true until after op becomes active.
active (op)	is true when the access defined by op takes place.
wait-term (op)	becomes true after access op is completed and remains true until after the termination phase begins.
term (op)	is true when termination phase of op is completed.

The truth values of the predicates are depicted in Fig. 1.

Specifications in SYSL will involve these predicates as well as predicates on the resource state. The model does not introduce any major restrictions on the class of synchronization problems that can be solved, or the mechanisms that can be used for synchronization, but are motivated by a desire to achieve a suitable abstraction of synchronization. Structured mechanisms for achieving synchronization such as monitors [14], serializers [3], sentinels [17], and synchronizing resources [2] fit this model.

Authorized licensed use limited to: to the Claremont Colleges. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

III. The Specification Language

This section is devoted to the development of SYSL—a language for specifying the synchronization properties of concurrent processes. The temporal logic based specification language facilitates expressing synchronization properties while satisfying the following desirderata.

- The language should be capable of specifying synchronization constraints of concurrent programs.
- The language should have intuitive interpretations as well as unambiguous semantics.
- The language should be modular, so that distinct properties are specifiable independent of each other.

Said differently, *human engineering* considerations are as consequential as considerations of *expressive completeness.*

The strategy espoused by this paper is as follows. Since, in our experience, there appears to be only a limited number of *characteristic* properties affecting the synchronization of concurrent processes, the formal statements of these properties can be *packaged* by employing appropriate keywords. For instance, suppose execution of all operations in class A have to exclude execution of operations in class B. This fact can be expressed by a statement involving the names A and B, and the keyword *excludes.* This statement is to be endowed with the semantics of the temporal logic expression that asserts exclusion.

This section shows the development of the specification language in the following way. We begin with a discussion of the parameters of synchronization. Then we introduce constructs for specifying characteristic properties affecting synchronization. In a concurrent environment, quick response of the synchronizer to individual requests is of predominant importance. This response is dependent on how fair the synchronizer is in servicing requests. Specification of fairness in SYSL is discussed in Section III-B2. In SYSL, each distinct property can be specified independent of the rest and hence specifications must be checked for their consistency. Detection of inconsistencies and other errors in a set of specifications is the subject of Section III-C. Specifications in SYSL have been used for a dichotomy of purposes: analyzing an extant synchronizer with respect to given specifications, and synthesizing code for a synchronizer that matches the given specifications. These two uses of SYSL specifications are discussed in Section III-D. Questions relating to completeness and human engineering aspects of SYSL are dealt with at the end of the paper.

We will use the *alarm clock problem* [14] to illustrate important aspects of SYSL. The alarm clock is a system facility that is shared by processes that need to be awakened after a specified time period. Executing the *wakeup* operation restarts a process. An argument to a wakeup request specifies when the process is to be awakened. Time is maintained by the alarm clock through the program variable "time." "Time" advances when a *tick* operation is executed. (Henceforth time will refer to the time maintained by the alarm clock.)

In the following description of SYSL constructs, "expr" enclosed within braces as in {expr} stands for an arbitrary number of textual occurrences of "expr."

A. Parameters of Synchronization

We distinguish between the terms *operation* and *operation-class.* This distinction is typified by the following example: in the alarm clock, two operation-classes are involved, *wakeup* and *tick.* A wakeup (tick) operation is an instance of the operation-class wakeup (tick). Every tick operation increments time by one. A wakeup operation is serviced at the end of the time period specified with the request for the operation.

Type of Requested Access: In general, constraints on servicing an operation will depend upon various characteristics of the operation, for example, how the operation modifies the state of the resource. Evidently, the synchronizer should be cognizant of the identity of individual operations.

The *operation classes* declaration lists the names of various classes of operations that a given synchronizer is expected to service as in

OPERATION-CLASSES {<operation-class>};

In some problems, a set of operation classes possess exactly the same specifications. A natural way of specifying such problems is to consider one such class and indicate that the specifications for that class are applicable to all operation classes in a given *set* of operation classes.

In a similar vein, since all operations in a particular class must satisfy similar synchronization constraints, it should be sufficient to specify constraints for a single operation in that class. Hence we permit universal quantification over operations in a given class.

Arguments to Operations: The specification language should permit declaration of the argument(s) to an operation and provide some means to refer to these arguments in the course of specifying constraints. Arguments to operations in a particular class can be specified by extending the declaration of operation classes as in

OPERATION-CLASSES
{<operation-class> (<argument> {, <argument>});}

The construct

<operation>. <argument>

refers to the argument of the named operation.

Example: Consider the following declaration for the alarm clock problem:

OPERATION-CLASSES tick;
wakeup(interval);

This declares that

- *tick* and *wakeup* are two operation classes, and
- *interval* is the argument to a *wakeup* operation.

Arguments need be specified only when their value is required for synchronization. If arguments are included, specifications can be constrained to apply only to operations whose arguments satisfy certain requirements. If the identity of a calling process affects synchronization, it can be included as an argument.

State of the Shared Resource: This is an important factor affecting synchronization. In particular, there may be a need to know the state of a shared resource either when the request for an operation arrives or is serviced. For example, in the alarm clock problem, *time* constitutes the state of the alarm clock and determines *when* a wakeup operation is serviced. Resource state variables are declared using the following

Authorized licensed use limited to: IEEE Xplore. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

specification:

RESOURCE-STATE-SPECIFICATION
{<resource-state-var> INITIALLY <initial-value>}.

Example:

RESOURCE-STATE-SPECIFICATION
time INITIALLY 0

declares that *time* is the resource state. The clause INITIALLY **0** provides information required for initializing the alarm clock. The specification language facilitates referring to the state of a resource at the beginning and end of each phase of an operation. This is made possibe by *qualifying* the resource state name by the point of interest as in

time AT request|a

which returns the value of time at the beginning of operation a's request phase. The general form of this construct is

<resource-state-var> <where> <phase>|<operation>.

<where> could be AT or AFTER.

State of the Synchronizer: As opposed to the parameter just described, the state of the synchronizer refers to the status of the synchronized operations. In general, actions of a synchronizer can be affected by

- the operations waiting for service,
- the operation being serviced,
- the active operations, and
- the termination of operations

and hence the predicates request, service, active, and term may be involved in the specifications.

The above parameters of synchronization appear in SYSL constructs to be introduced next.

B. Language Constructs

SYSL is a high-level specification language in which properties that are normally relevant to synchronization are expressible in a readable form. Statements in SYSL can be considered to be macros with appropriate definitions in terms of temporal logic statements. This section describes the constructs in this language. Broadly speaking, synchronization depends on

1) the properties of the resource being shared,
2) the properties of the operations on the resource, and
3) the expected response.

This categorization is not meant to imply that these factors are mutually exclusive. Often since resource state is modified by the operations on the resource, the interaction of factors 1) and 2) is vital to the servicing of operations.

Questions relating to the responsiveness of the synchronizer, a factor which determines the performance of concurrent processes, are usually neglected in the specification and construction of synchronization code. One often comes across the statement, "if there are multiple requests waiting for service, the synchronizer will handle the requests with fairness," with the statement within quotes left unexplained. The subject of responsiveness will be developed further in the next subsection.

1) *Properties of a Resource and Operations on the Resource*: *State Invariant of the Shared Resource*: A very important property which affects synchronization is the state-invariance of a resource. By this we mean those aspects of a resource which should always hold. For instance, given a pool of resources, the synchronizer should service allocate and free requests from user processes such that the number of resources allocated is always less than or equal to the maximum number of resources in the pool. In general, such requirements are specified as

ALWAYS <resource-state-invariant>

where <resource-state-invariant> is a predicate on resource state variables.

Resource State Changes: During the execution of an operation, the *state* of the shared resource may be altered; for instance, in an alarm clock, *tick* increments *time*. In SYSL this would be specified as

FOR EACH t IN Tick: time <- time + 1.

The above specification has the following semantics:

∀t∈tick,
□ {term (t) => [time = (time AT service|t) + 1]}

This expresses the fact that a tick operation increments time by 1. As we shall see, the fact that *ticks* execute in exclusion will have to be specified independently. The abstract syntax for the above specification is

FOR EACH op IN <operation-class>:
<resource-state-var> <- <newvalue>

<Newvalue> is an expression that involves values of the <resource-state-var> when the request for op arrives or when op is serviced, or arguments to op. Any resource state variable which appears in <newvalue>, unless explicitly qualified, refers to the value of the resource state variable when op is serviced.

In addition, SYSL also permits the specification of changes to resource state due to the execution of individual phases. SYSL syntax for such specifications is

FOR EACH op IN <operation-class>:
IF <predicate1> AT <phase>|op
THEN <predicate2> AFTER <phase>|op

where <predicate1> and <predicate2> are predicates on the resource state and may involve arguments to "op." Thus the phase specified in <phase> changes the resource state such that if <predicate1> holds at the beginning of the phase then <predicate2> holds at the end of that phase. The semantics of the above statement is

∀op∈<operation-class>,
□ {(at (<phase>|op) ∧ <predicate1>)
=>
□ (after (<phase>|op) => <predicate2>)}

Mutual Exclusion of Operations: Each tick operation increments time by one. Clearly, for the alarm clock to maintain the correct time, two tick operations should not be concurrently active. This exclusion between tick operations is specified in SYSL as

tick OPERATIONS EXCLUDE EACH OTHER.

Authorized licensed use limited to: Claremont Colleges. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

This specification has the following semantics:

$$\forall t1, t2 \in tick, t1 \neq t2, \quad \Box \neg\{active(t1) \wedge active(t2)\}.$$

The general syntax for exclusion between operations in a class is

<operation-class> OPERATIONS EXCLUDE EACH OTHER.

The other type of exclusion occurs between operations in different classes. The temporal assertion

$$\forall op1 \in OPC1, \forall op2 \in OPC2, \quad \Box \neg\{active(op1) \wedge active(op2)\}$$

expresses exclusion between executions of operations in OPC1 and those in OPC2.

OPC1 OPERATIONS EXCLUDE OPC2 OPERATIONS

is the corresponding SYSL specification.

For example, since execution of a wakeup operation depends on *time*, a variable changed by *tick*,

tick OPERATIONS EXCLUDE wakeup OPERATIONS

is a necessary specification for the alarm clock problem.

If there is no specification requiring the exclusion of two operations then they can be concurrently active.

Operation Sequences: There are synchronization situations in which a set of operations are required to always execute in a strict sequence. By a sequence we mean that some operation is to be serviced only after some other operation is serviced. When a particular set of operations is to execute in a fixed sequence, it is useful to be able to specify this explicitly rather than in terms of necessary conditions for servicing operations.

If operations of a particular class are numbered in the order in which they are serviced, then by Ai we refer to the ith temporal instance of the operation class A. (We are numbering instances of operations for the sole purpose of giving the semantics of sequences.) The ith execution of a sequence S consists of execution of ith instance of each operation class belonging to the sequence S.

Suppose in a sequence S, operations in class A follow those in class B. Then

$$\forall i \in \{\text{natural number}\}, \quad [service(Ai) \text{ ONLYAFTER } service(Bi)]$$

One would specify this sequence in SYSL as

OPERATION SEQUENCE S: A FOLLOWS B.

Such specifications express the order in which operations can be serviced. Here, the number of A's serviced is never more than the number of B's serviced.

It is possible for two instances of a sequence to execute concurrently or in exclusion. If a sequence *excludes itself* then the (i + 1)th execution of the initial operations of the sequence can be serviced only after the termination of the ith execution of the terminal operations of the sequence. Exclusion of a sequence can be specified in the same way operation exclusion is specified.

Notice that these specifications sequence only the *servicing* of operations. That is, operations within a sequence can execute concurrently if other specifications permit. So if operations within a sequence should exclude, there have to be additional specifications so stating.

Service-Constraints: These are the specifications of explicit conditions under which an operation can be serviced. Since exclusion, resource-state-invariance and other high-level properties can be expressed independently, a service-constraint specification need exist only if constraints not implied by other specifications are required.

The SYSL specification

SERVICE-CONSTRAINT
FOR EACH op IN <operation-class>:<predicate1>

requires <predicate1> to be true when op is serviced. This is equivalent to the following temporal logic assertion

$$\forall op \in \langle operation\text{-}class \rangle, \quad \Box\{service(op) \Rightarrow \langle predicate1 \rangle\}.$$

Example: In the alarm clock problem, the interval specified by the argument to the operation should have elapsed when a wakeup operation is serviced, i.e.,

SERVICE-CONSTRAINT
FOR EACH w IN wakeup:
time = ((time AT request|w)+w.interval)

Here we notice two language constructs at work:

(time AT request|w) stands for the value of time when request for w arrives

w.interval stands for the argument to w.

One of the required constraints for an operation to be serviced is that there be a request for the operation. Since this is implied by the semantics of predicate "service," this constraint need not be specified.

It is not always the case that all operations in a class have the same set of constraints. Constraints could depend on the arguments to a request or on the state of the resource at the time a request arrives. To facilitate specifications such as these, the above statement can be extended as follows:

SERVICE-CONSTRAINT
FOR EACH op in <operation-class>:
IF <predicate1> AT request|op THEN <predicate2>

which specifies the following:

$$\forall op \in \langle operation\text{-}class \rangle, \quad \Box\{\langle predicate1 \rangle \wedge at\,(request|op) \Rightarrow \Box\{service(op) \Rightarrow \langle predicate2 \rangle\}\}$$

whereby if <predicate1> holds at the beginning of the request phase of op, then <predicate2> is a constraint for servicing op. In general, <predicate1> and <predicate2> above are predicates on resource state variables and arguments to the operations.

The specifications of mutual exclusion, sequencing, resource-state-invariant, and service-constraints determine the necessary conditions for servicing an operation. We say that *enabled* (*op*) is true if necessary conditions for servicing operation op are satisfied. In Section III-D we discuss some of the transformation rules for determining enabled (op) for each synchronized operation op.

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:32 from IEEE Xplore. Restrictions apply.

In our language, by convention, priority specifications do not contribute to the necessary conditions. Priority specifications are used only to order operations that have satisfied their necessary conditions.

Priority: In general, when we say that an operation (say b) has higher priority than another (say a), we mean that a can be serviced only after b, i.e.,

$$\Box\{[\mathrm{req}(a) \wedge \mathrm{req}(b) \wedge (\mathrm{pr}(a) < \mathrm{pr}(b))] \Rightarrow [\mathrm{service}(a)\ \textsc{onlyafter}\ \mathrm{service}(b)]\}$$

where pr (a) stands for the priority of operation a. Of course, whenever either a or b is serviced, it must be enabled. The priority we are discussing here belongs to the genre of non-preemptive priority. Thus, once serviced, an operation is permitted to complete execution even if a higher priority operation arrives in the meantime.

We classify priority into the following two categories:

1) priority within requests of a particular operation class, otherwise known as *intraclass priority* and

2) priority between different operation classes, otherwise known as *interclass priority*.

We consider the former category first. Normally, priority between operations in a particular class is given by some rule (*expr* in the following expression) and evaluating the rule for each operation gives the priority for that operation.

$$\forall \mathrm{op1}, \mathrm{op2} \in \langle\text{operation-class}\rangle,\ \Box\{[\mathrm{req}(\mathrm{op1}) \wedge \mathrm{req}(\mathrm{op2}) \wedge (\mathrm{expr}(\mathrm{op1}) < \mathrm{expr}(\mathrm{op2}))] \Rightarrow [\mathrm{service}(\mathrm{op1})\ \textsc{onlyafter}\ \mathrm{service}(\mathrm{op2})]\}$$

where expr(a) stands for the value of expr evaluated for operation a.

In the case of interclass priority, operations in a class having a lower priority can be serviced only after all operations in higher priority classes have been serviced. Thus, if class OPC2 has a priority higher than OPC1, then

$$\forall \mathrm{op1} \in \mathrm{OPC1}, \forall \mathrm{op2} \in \mathrm{OPC2},\ \Box\{[\mathrm{req}(\mathrm{op1}) \wedge \mathrm{req}(\mathrm{op2})] \Rightarrow [\mathrm{service}(\mathrm{op1})\ \textsc{onlyafter}\ \mathrm{service}(\mathrm{op2})]\}.$$

So far we have assumed that the priority of a particular operation is static. In general however, both interclass and intraclass priorities can depend on resource state. This dependence can be specified in SYSL through the use of *resource-state-predicates* which are predicates on the state of the resource. Using resource-state-predicates, it is possible to specify resource state dependent priority rules and priority relationships between operation classes.

Specification:

```
INTRACLASS PRIORITY AMONG REQUESTS
    <operation-class> : <priority-rule>
        WHEN <resource-state-predicate>
```

Informal Semantics:

If "OP : expr WHEN r" is an intraclass priority specification, then expr gives the priority rule applicable to operations in class OP when the resource state satisfies r.

Formal Semantics:

$$\forall \mathrm{op1}, \mathrm{op2} \in \mathrm{OP},\ \Box\{[r \wedge \mathrm{req}(\mathrm{op1}) \wedge \mathrm{req}(\mathrm{op2}) \wedge (\mathrm{expr}(\mathrm{op1}) < \mathrm{expr}(\mathrm{op2}))] \Rightarrow [\mathrm{service}(\mathrm{op1})\ \textsc{onlyafter}\ \mathrm{service}(\mathrm{op2})]\}$$

where, as before, expr(a) stands for the value of expr evaluated in the context of req(a). Now we turn our attention to specifying interclass priority in SYSL.

Specification:

```
INTERCLASS PRIORITY AMONG REQUESTS
    <operation-class-2> > <operation-class-1>
        WHEN <resource-state-predicate>
```

Informal Semantics:

Given an inter-class priority statement "OPC2 > OPC1 WHEN r," if current resource state satisfies r, then operations in class OPC2 have higher priority than those in OPC1.

Formal Semantics:

$$\forall \mathrm{op1} \in \mathrm{OPC1}, \forall \mathrm{op2} \in \mathrm{OPC2},\ \Box\{[r \wedge \mathrm{req}(\mathrm{op1}) \wedge \mathrm{req}(\mathrm{op2})] \Rightarrow [\mathrm{service}(\mathrm{op1})\ \textsc{onlyafter}\ \mathrm{service}(\mathrm{op2})]\}.$$

Priority specifications without the WHEN clause are applicable under all resource states.

Another classification of priority is based on whether it applies to all requests or only to those which have satisfied their necessary conditions. So far we have been discussing the first category. In the latter category, an operation can be serviced only after servicing enabled operations with higher priority. Formal semantics in this case is obtained by substituting *enabled(op)* for *req(op)* in the definitions above.

Example: In the alarm clock problem, we need to ensure that all eligible processes are awakened before *time* is incremented. So we have

```
INTERCLASS PRIORITY AMONG ENABLED OPERATIONS
    wakeup > tick.
```

This ensures that requests for ticks are not serviced while enabled wakeup operations exist.

Policies adopted in operating systems for process scheduling [5] are based on the priority of the processes and the system state. Normally, it is not essential for the server of a shared resource to know the identity of the customer processes. However, in situations where servicing of operations takes place based on the priority assigned to individual processes, the following simple strategy can be employed: the priority of a process is passed as an argument to the requested operation and the priority specification for the server expressed in terms of this argument.

2) *Specification of Fairness*: These specifications express the required behavior of the synchronizer so that no operation is unduly delayed. We give below various versions of such fairness specifications [25]. One chooses a fairness criterion appropriate for the problem being specified.

Fairness-1: One way of servicing operations is by considering them in the order of arrival of their requests. The earliest

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

to arrive will always be chosen for service. Formally, the expression

∀op1, op2∈<operation-class>,
 □{[at(request|op2) AFTER at(request|op1)]
  =>
  [service(op2) ONLYAFTER service(op1)]}

states that if the request phase of op2 began after that of op1 then op2 can be serviced only after op1. Thus requests for operations in <operation-class> are serviced in the order of their arrival. In SYSL, such a requirement would be expressed as

SERVICE EACH op IN <operation-class>
 FIRST-COME-FIRST.

However, fairness-1 may not always be consistent with the rest of the specifications, especially if the operations have dependent constraints.

More general are situations where the holding of some condition implies the eventual servicing of some operation op, that is,

<expression> => ◇service(op)

where <expression> is an assertion dependent on enabled(op).

Given the existence of various fairness criteria, the obvious issue to be examined is: under what circumstances is a particular fairness criterion admissible? A given type of fairness is said to be *admissible* if some synchronizer for the problem can guarantee such a fairness. As we introduce various fairness criteria, we also examine their admissibility. In what follows, *predicate(op)* involves no temporal operators.

Fairness-2: A strong form of fairness is

∀op∈<operation-class>,
 □{predicate(op) => ◇service(op)}.

Thus once *predicate(op)* holds, op has to be eventually serviced. In SYSL this would be expressed as

SERVICE EACH op IN <operation-class>
 IF predicate(op) IS TRUE.

Consider the following situation:

> Suppose fairness-2 is specified for two operations, say *op2* and *op1*. Assume that both operations are enabled and that predicate(op1) and predicate(op2) hold. Suppose op1 and op2 are such that serving any one of them will permanently disable the other.

In this case, Fairness-2 will not be admissible. The truth of one of the following is sufficient to show the admissibility of this fairness to an operation op1.

- When predicate(op1) is true, no other operation is enabled.
- Even if another operation op2 is enabled, enabled(op1) remains true until op1 is serviced.
- Even if another operation op2 is serviced, enabled(op1) will eventually become true again.

Fairness-3: One way to weaken the previous version of fairness is by requiring that an operation be eventually serviced if the predicate holds *repeatedly*, that is,

∀op∈<operation-class>,
 □{□◇predicate(op) => ◇service(op)}.

Here it will suffice if predicate(op) is enabled infinitely often, i.e., repeatedly, for op to be serviced. This would be specified in SYSL as

SERVICE EACH op IN <operation-class>
 IF predicate(op) IS REPEATEDLY TRUE.

When priority specifications are present, only a still weaker form of fairness is admissible. The sequential model assumed for the synchronizer precludes the immediate recognition of enabled operations. This implies that although at a given time an operation may be eligible for service, a request for a higher priority operation may have arrived before the synchronizer recognizes this fact, thus preventing the synchronizer from servicing the former. Hence we have the following type of fairness which is typically used for lower priority operations.

Fairness-4:

∀op∈<operation-class>,
 □{(predicate(op) UNTIL service(op)) => ◇service(op)}.

This states that if predicate(op) remains true until op is serviced, then op should be eventually serviced. This essentially means that predicate(op) *remain* true (as opposed to being repeatedly true) till the sychronizer recognizes it and takes appropriate action. The corresponding SYSL specification is

SERVICE EACH op IN <operation-class>
 IF predicate(op) IS ALWAYS TRUE.

Example: In the alarm clock problem, we require that once enabled, every *wakeup* operation be eventually serviced. Tick requests are required to be serviced in the order of their arrival. Also, due to its lower priority, a tick operation is required to be serviced according to fairness-4:

SERVICE EACH w IN wakeup IF enabled(w) IS TRUE.
SERVICE EACH t IN tick FIRST-COME-FIRST.
SERVICE EACH t IN tick
 IF [enabled(t) ∧ FOR EACH w in wakeup, ¬enabled(w)]
  IS ALWAYS TRUE.

The current version of SYSL permits only the above four versions of fairness. The implications of using arbitrary temporal logic statements for expressing fairness are being investigated from the viewpoint of readability and synthesis.

Given below is the overall specification for the alarm clock problem. Note that the specification for the problem is the conjunction of the specifications of individual properties.

SYNCHRONIZER Alarm-Clock IS
OPERATION-CLASSES Tick;
        Wakeup(interval);
RESOURCE-STATE-SPECIFICATION
 time: INITIALLY 0;
RESOURCE-STATE-CHANGES
 FOR EACH t IN tick: time <- time+1;
EXCLUSION
 Tick OPERATIONS EXCLUDE wakeup OPERATIONS;
 Tick OPERATIONS EXCLUDE EACH OTHER;
SERVICE-CONSTRAINTS
 FOR EACH w IN wakeup:
  time = ((time AT request|w) + w.interval)
INTERCLASS PRIORITY AMONG ENABLED REQUESTS
 wakeup > tick;

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

```
FAIRNESS
   SERVICE EACH w IN wakeup IF enabled(w) IS TRUE
   SERVICE EACH t IN tick FIRST-COME-FIRST
   SERVICE EACH t IN tick
      IF [enabled(t) ∧ FOR EACH w in wakeup,
         ¬enabled(w)] IS ALWAYS TRUE.
```

But for the dependence of fairness and priority on the remaining specifications, each distinct property of the problem is specified independently of the rest. This attests so the modularity and ease of use of SYSL. In the next subsection we discuss the detection of inconsistencies between independent specifications.

As an additional example of use of this specification language, let us specify the behavior of a resource manager which manages a fixed number (say 10) of similar resources. User processes acquire a resource by executing the operation *allocate*, and release the resource by executing the operation *free*. The number of resources free at any given time is maintained by *avail*. *Maxavail* gives the maximum number of available resources. To expedite release of resources, free is given priority over allocate. This is a typical problem which arises in the context of resource management and involves resource state, and changes to resource state. Given below is the SYSL specification for the resource manager.

```
SYNCHRONIZER Resource-Manager IS
OPERATION-CLASSES Allocate;
                  Free;
RESOURCE-STATE-SPECIFICATION
   maxavail: CONSTANT 10;
   avail: INITIALLY maxavail;
RESOURCE-STATE-CHANGES
   FOR EACH a IN allocate: avail <- avail - 1;
   FOR EACH f IN free: avail <- avail + 1;
RESOURCE-STATE-INVARIANCE
   ALWAYS {(avail ≤ maxavail) ∧ (avail ≥ 0)};
OPERATION EXCLUSION
   Allocate OPERATIONS EXCLUDE Free OPERATIONS;
   Free OPERATIONS EXCLUDE EACH OTHER;
   Allocate OPERATIONS EXCLUDE EACH OTHER;
INTERCLASS PRIORITY AMONG ENABLED OPERATIONS
   Free > Allocate;
FAIRNESS
   SERVICE EACH f IN free IF nec-cond(f) IS TRUE
   SERVICE EACH f IN free FIRST-COME-FIRST
   SERVICE EACH a IN allocate FIRST-COME-FIRST
   SERVICE EACH a IN allocate
      IF [nec-cond(a) ∧ FOR EACH f IN free, ¬nec-cond(f)]
         IS ALWAYS TRUE.
```

In [28] several standard synchronization problems have been specified in SYSL.

C. Erroneous Specifications and Their Detection

In SYSL, distinct properties are specified independently of each other. This permits users to focus on one property at a time. However, this could result in inconsistent, incomplete, or deadlock-prone specifications. Here we briefly outline techniques for detecting the presence of such errors. Further details may be found in [28].

Specifications may be *inconsistent* if no synchronizer can have the required behavior. For instance, interclass priority specifications may be such that on taking the transitive closure of the priority relationship between operation-classes, one may find that an operation-class has higher priority than itself, thereby revealing an inconsistent set of priority specifications. A similar form of inconsistency can arise in sequence specifications also. Inconsistencies may also occur in the specification of fairness. In Section III-B, we introduced the notion of admissibility of a fairness criterion in order to detect inconsistent fairness specifications.

A set of specifications is said to be *incomplete* if further specifications are required to completely specify the problem. For example, in the alarm clock problem, since *tick* increments time, it is necessary to specify the exclusion between tick operations and between tick and wakeup operations. In general, only a specifier can answer questions regarding the completeness of a set of specifications since two distinct sets of specifications may be specifying two different synchronization problems.

A set of specifications for a synchronization problem is *deadlock-prone* if processes that access a shared resource controlled by a synchronizer are liable to deadlock. It is possible to give sufficient conditions for the *absence of deadlock* among the processes being synchronized. For this purpose, we consider each subset of the set of operation-classes. Processes requesting operations in that subset will not deadlock if at least one such request can be serviced. To show this to be the case, we prove that the disjunction of the necessary conditions of the operations in the subset is always true. For this purpose, we make use of the temporal order in which user processes access the resource.

Algorithmic detection of erroneous specifications is, in general, difficult. However, some of these error detection techniques have been built into our synthesis system [28] such that errors are detected during preanalysis of the specifications and yet others as synthesis progresses.

D. Use of SYSL Specifications

Specifications in SYSL serve a dichotomy of purposes: synthesis and analysis of synchronization code. Now we briefly discuss how each is accomplished.

Synthesis: Synchronization properties expressed in SYSL serve as input to a system which automatically generates code for a synchronizer that guarantees the specified properties [28]. This system uses meaning-preserving transformation rules to transform input specifications into code for a synchronizer. Hence the resulting code guarantees the specified properties thus eliminating the need for verification.

The first task is to determine the necessary conditions for servicing each synchronized operation. This is done via transformation rules such as the following.

The Mutual Exclusion Transformation Rule: This rule specifies sufficient conditions for servicing operations in order to satisfy mutual exclusion requirements:

```
∀op1, op2,
  {□ {service(op1) => ¬[active(op2) OR wait-active(op2)]} ∧
   □ {service(op2) => ¬[active(op1) OR wait-active(op1)]}}
   =>
      □ ¬{active(op1) ∧ active(op2)}
```

Note that the consequent of this rule expresses the exclusion between op1 and op2.

Authorized licensed use limited to: the Claremont Colleges. Downloaded on [illegible] at 07:52 from IEEE Xplore. Restrictions apply.

The Priority Transformation Rule: Priority specifications will be satisfied if an operation is serviced only when no requests for operations of higher priority are present.

$$\begin{aligned}&\forall \text{op1}, \text{op2}, \text{op1} \neq \text{op2},\\ &\Box\{\text{service(op1)} \Rightarrow \neg\text{req(op2)}\}\\ &\quad\Rightarrow\\ &\Box\{[\text{req(op1)} \wedge \text{req(op2)}] \Rightarrow\\ &\quad[\text{service(op1) } \textsc{onlyafter} \text{ service(op2)}]\}.\end{aligned}$$

The consequent of this rule expresses the priority for op2 over op1.

The constraints on servicing an operation is the conjunction of 1) the constraints derived by the application of the transformation rules, 2) the implicit requirement that

$$\forall \text{op}\ \Box\{\text{service(op)} \Rightarrow \text{req(op)}\},$$

and 3) the service-constraint specifications.

Code is synthesized such that necessary conditions of requests are evaluated and appropriate actions taken so as to satisfy the specified fairness. Details of the synthesis algorithm can be found in [28]. Manna and Wolper [22] as well as Clarke and Emerson [8] have also attempted the synthesis of synchronizing processes starting from temporal logic specifications. The differences between our approach and the above arise from the human-engineering features of SYSL. In addition, while Manna and Wolper use temporal logic to specify synchronizing processes by abstracting concurrent computation to sequence of events and Clarke and Emmerson utilize branching time logic for specifying synchronizing processes, our specification language is based on linear time temporal logic and utilizes the notion of states.

Analysis: When analysis of synchronizers is undertaken, the conditions imposed by the synchronizer are first determined by deriving the conditions that exist when an operation is serviced. These should imply the necessary conditions obtained from the specifications using the transformation rules above. Fairness is proven using the control flow properties of the synchronizer. Details of this method can be found in [26].

IV. Summary and Evaluation

We have used the formalism of temporal logic to precisely define synchronizers for concurrent processes. This has been done through the introduction of an operational model for a synchronizer. Most structured mechanisms for synchronization fit this model.

We have systematized and abstracted features of synchronization control into a set of language constructs based on temporal logic, which seems to be a natural tool to express both safety and liveness properties of concurrent systems. Use of temporal constructs such as *always*, *eventually*, and *until* along with those constructs derivable from them, result in intuitive specifications for synchronization problems. It concentrates on the human-engineering features such as:

- properties usually relevant to synchronization, e.g., mutual exclusion, resource-state-invariance, sequence and priority are specified through high-level constructs which have intuitive interpretations and precise semantics, and
- users need concern themselves with only one aspect of the problem at a time since each property can be expressed independent of the rest.

In practice, there may be a wide gap between the informal notion of a synchronization problem and the corresponding formal specification. Hence, often one is not sure of the correctness of the specifications themselves. We believe that our approach to specifying synchronization via SYSL is a step towards bridging this gap.

As we summarize our temporal logic based specification language, we shall evaluate it against our own requirements as well as compare it with existing specification methods.

Precise Semantics: Since precise semantics was one of the underlying design requirements, every high-level specification statement in the language has formal temporal semantics. Temporal logic has been used for specifying concurrent systems [4], [23] for specifying protocols in distributed systems [13], [30], and for specifying synchronizing processes [8], [22].

As regards other proposals, expression based languages [31] are attractive due to their denotational orientation. Of these, path expressions [12] are perhaps the most widely referenced. However, due to the proliferation of various versions of path expressions [1], [16], [20], it is not clear whether any single implementation of path expressions conforms to any proposed formal semantics [6], [19].

Due to the inability of the other languages to express liveness properties, *fairness* is often expressed in ambiguous and informal terms such as

> every request should be serviced.

As we have demonstrated earlier (Section III-B), no single fairness criterion is applicable to all synchronization situations. So such statements are essentially unsatisfiable and due to their informality, unverifiable for a given system. Our language, due to its ability to specify eventual behavior, is capable of expressing a variety of responsiveness criteria.

Expressiveness: This measure relates to whether a specification language can express *any* synchronization situation. Obviously, it is not possible to answer this question in absolute terms due to the absence of a composite list of synchronization situations.

Following [29], we conjecture that conditional critical regions in which the conditions and assignments are linear expressions involving shared variables is adequate to code synchronization problems. These conditions and actions are expressible in our language by means of specifications of service-constraints and state changes (Section III-B). Clearly then, our language is complete in this sense. In addition, users directly specify higher level properties such as mutual exclusion of operations since SYSL permits the specification of properties themselves as opposed to the conditions that they imply. Transformation rules of the synthesis system synthesize the conditions for each critical region.

Considering path expressions once again, since sequencing of operations has been considered as their chief property, in preference to exclusion or resource-state-invariance, to specify the latter, the influence of exclusion or invariance on the ordering of operations has to be first determined. Needless to say, this is a nontrivial exercise. In Greif's approach to the specification of synchronization [9], synchronization requirements are specified as partial orderings of *key* events pertaining to an operation. Laventhal [21] utilizes this approach to specify the

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

properties of synchronizers. Here again, properties such as mutual exclusion have to be specified in terms of orderings of the key events.

Modifiability: This factor relates to the ease with which modifications can be incorporated in a set of specifications. It also concerns the modularity and extensibility of the specifications. For instance, suppose it is found necessary to extend the specifications for a synchronization problem by giving priority to a class of operations.

- In languages where explicit conditions for servicing operations are to be specified [10], it is necessary to determine the influence of the added priority on servicing operations.
- In Laventhal's language, specified orderings of key events have to be modified, or further constrained, to take priority into consideration.
- In some versions of path expressions, the priority operator can be used for this purpose.
- In our language, since priority is specified as a distinct property, such an addition would mean an additional specification statement, without regard to other specifications (as long as the added priority specification does not contradict another priority or fairness specification).

Since each property is specified independently of others, our specification language produces readable, modular, and extensible specifications.

Ease of Use: This relates to the facility with which a user can specify in this language, which admittedly is a subjective factor. Nevertheless, it can be said without any apprehension of being contradicted, that the closer the statements in a language are to a human conceptualization of the properties being specified, the easier it is to use the language. Certain germane points have already been made in this regard during the discussion of expressiveness and modifiability.

SYSL has been designed by abstracting properties of interest through keywords that correspond to the property being specified. This, in addition to the modular nature of the specifications makes it intuitively appealing and hence, easy to use. On the other hand, current specification languages, due to their lack of facilities to specify relevant properties in a modular, natural manner, often produce contrived specifications.

Another criterion which determines the ease of use of a given specification language concerns the details a user is expected to specify. Let us consider an example to clarify this criterion. In the notation used in [7] and [10], mutual exclusion is expressed through the specifications of

- all distinct states of a shared resource,
- preconditions for each type of access, and
- state transitions corresponding to each access.

In SYSL, a user is expected to specify state changes only if they affect synchronization. High-level properties such as mutual exclusion are specified not through the specifications of state changes but using a specification involving the name(s) of operation classes and the keyword "excludes," thereby conveying a greater degree of abstraction than state transition based specifications.

A closely related issue is the notion of correctness of the specifications. In practice, specifications tend to be either totally informal or very formal. The first suffers from ambiguities and hence users could be unsure of whether a specification stands for what they want to specify. In the latter case, due to the conceptual gap between users' view of the problem and the formal specifications, it is not obvious to casual users whether their specifications are correct. The statements in our specification language provide a semblance of informality while having precise semantics. Also, we provide techniques for the detection of errors in SYSL specifications.

Acknowledgment

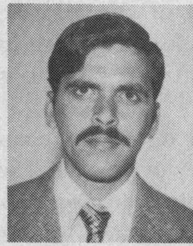
Our thanks to the referees whose constructive remarks considerably influenced the present form of the paper.

References

[1] S. Andler, "Predicate path expressions," in *Proc. 6th Annu. Symp. POPL*, Jan. 1979, pp. 226–236.
[2] G. R. Andrews, "Synchronizing resources," *ACM Trans. Programming Languages Syst.*, vol. 3, pp. 405–430, Oct. 1981.
[3] R. R. Atkinson and C. E. Hewitt, "Specification and proof techniques for serializers," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 10–23, Jan. 1979.
[4] M. Ben-Ari and A. Pnueli, "Temporal logic proofs of concurrent programs," Tel Aviv Univ., Tech. Rep., Nov. 1980.
[5] A. J. Bernstein and J. C. Sharp, "A policy-driven scheduler for a time-sharing system," *Commun. Ass. Comput. Mach.*, vol. 2, pp. 74–78, Feb. 1971.
[6] V. Berzins and D. Kapur, "Denotational and axiomatic definitions for path expressions," Massachusetts Inst. Technol., Comput. Structures Group Memo. 153-1, Nov. 1977.
[7] P. Brinch-Hansen and J. Staunstrup, "Specification and implementation of mutual exclusion," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 365–370, Sept. 1978.
[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop Logics of Programs* (*Springer-Verlag Lecture Notes in Comput. Sci.*), vol. 131, 1981.
[9] I. Greif, "A language for formal problem specification," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 931–935, Mar. 1977.
[10] P. Griffiths, "SYNVER: A system for the automatic synthesis and verification of synchronization processes," Harvard Univ., TR Tech. Rep. 22-74, 1974.
[11] V. Guttag, E. Horowitz, and D. Musser, "Abstract data types and software validation," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 1048–1064, Dec. 1978.
[12] A. N. Habermann, "Path expressions," Carnegie-Mellon Univ., June 1975.
[13] B. T. Hailpern and S. Owicki, "Verifying network protocols using temporal logic," in *Proc. NBS/IEEE Conf. Trends and Applications on Comput. Network Protocols*, 1980.
[14] C.A.R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 540–557, Oct. 1974.
[15] ——, "Communicating sequencial processes," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 666–677, Aug. 1978.
[16] B. Jayaraman and R. M. Keller, "Resource expressions for applicative languages," in *Proc. Int. Conf. Parallel Processing*, June 1980, pp. 160–167.
[17] R. M. Keller, "Sentinels: A concept for multiprocess coordination," Univ. Utah, UUCS-78-104, June 1978.
[18] L. Lamport, "'Sometime' is sometimes 'not never'," in *Proc. 7th Annu. Symp. POPL*, Jan. 1980, pp. 174–185.
[19] P. E. Lauer and R. H. Campbell, "Formal semantics of a class of primitives for coordinating concurrent processes," *Acta Informatica*, vol. 5, pp. 297–332, 1975.
[20] P. E. Lauer, P. R. Torrigiani, and M. W. Shields, "COSY–A system specification language based on paths and processes," *Acta Informatica*, vol. 12, pp. 109–158, Apr. 1979.
[21] M. S. Laventhal, "Synthesis of synchronization code for data abstractions," Massachusetts Inst. Technol., MIT/LCS/TR-203, June 1978.

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.

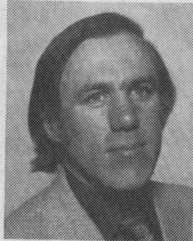
[22] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal logic specifications," in *Proc. Workshop Logics of Programs* (*Springer-Verlag Lecture Notes in Comput. Sci.*), vol. 131, 1981.
[23] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Programming Languages Syst.*, vol. 4, pp. 455-495, July 1982.
[24] A. Pnueli, "The temporal semantics of concurrent programs," in *Semantics of Concurrent Computation* (*Springer Lecture Notes in Comput. Sci.*), vol. 70, June 1979, pp. 1-20.
[25] —, "On the temporal analysis of fairness," in *Proc. 7th Annu. Symp. POPL*, Jan. 1980, pp. 163-173.
[26] K. Ramamritham and R. M. Keller, "Specifying and proving properties of sentinel processes," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 374-382.
[27] —, "On synchronization and its specification," in *Springer Lecture Notes in Comput. Sci.*, vol. 111, June 1981.
[28] K. Ramamritham, "Specification and synthesis of synchronizers," Ph.D. dissertation, Univ. Utah, Aug. 1981.
[29] H. A. Schmid, "On the efficient implementation of conditional critical regions and the construction of monitors," *Act Informatica*, vol. 6, pp. 227-249, 1976.
[30] R. L. Schwartz and P. M. Melliar-Smith, "Temporal logic specifications of distributed systems," in *Proc. 2nd Int. Conf. Distributed Syst.*, Apr. 1981.
[31] A. C. Shaw, "Software specification languages based on regular expressions," in *Proc. Software Tools Workshop*, May 1979, pp. 1-39.



Krithivasan Ramamritham received the B.Tech degree in electrical engineering and the M.Tech degree in computer science from the Indian Institute of Technology, Madras, India, in 1976 and 1978, respectively, and the Ph.D. degree in computer science from the University of Utah, Salt Lake City, in 1981.

Currently, he is an Assistant Professor in the Department of Computer and Information Science, University of Massachusetts, Amherst. His research interests include software engineering, operating systems and distributed computing.

Dr. Ramamritham is a member of the Association for Computing Machinery and the IEEE Computer Society.



Robert M. Keller received the B.S. and M.S.E.E. degrees from Washington University, St. Louis, MO, and the Ph.D. degree from the University of California, Berkeley.

Currently, he is a Professor of Computer Science at the University of Utah, Salt Lake City. From 1970-1976 he was an Assistant Professor of Electrical Engineering at Princeton University. His current research interests deal with numerous topics relating to multiprocessor implementations of functional languages, particularly using reduction and data-flow computation models.

Distributed Software System Design Representation Using Modified Petri Nets

STEPHEN S. YAU, FELLOW, IEEE, AND MEHMET U. CAGLAYAN, MEMBER, IEEE

***Abstract*—A model for representing and analyzing the design of a distributed software system is presented. The model is based on a modified form of Petri net, and enables one to represent both the structure and the behavior of a distributed software system at a desired level of design. Behavioral properties of the design representation can be verified by translating the modified Petri net into an equivalent ordinary Petri net and then analyzing that resulting Petri net. The model emphasizes the unified representation of control and data flows, partially ordered software components, hierarchical component structure, abstract data types, data objects, local control, and distributed system state. At any design level, the distributed software system is viewed as a collection of software components. Software components are externally described in terms of their input and output control states, abstract data types, data objects, and a set of control and data transfer specifications. They are interconnected through the shared control states and through the shared data objects. A system component can be viewed internally as a collection of subcomponents, local control states, local abstract data types, and local data objects.**

***Index Terms*—Control flow and data flow, design analysis, distributed software system, modified Petri net, software design representation.**

Manuscript received November 6, 1981; revised March 21, 1983. This work was supported by the U.S. Army Research Office under Contract DAA-C29-80-K-0092.

S. S. Yau is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

M. U. Caglayan was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the University of Petroleum and Minerals, Dhahran, Saudi Arabia.

I. Introduction

WE CONSIDER that a distributed computer system has a number of processing nodes connected by a message-based communication network. In addition to the physical distribution of hardware, conceptual distribution of both data and control is an essential characteristic of the system. To emphasize decentralized control, processing nodes will be highly autonomous in their availability, type of service they provide, their concern for protection of resources, and their reliability. The effects of actual choice of processing and communication hardware and system topology on design issues are not considered here.

The design of distributed software systems continues to be a challenging area of software engineering. A number of informal and formal design methods, which are primarily concerned with sequential software systems, have been proposed [1]-[5]. These methods do not directly address the design problems associated with parallel and distributed systems. Although

0098-5589/83/1100-0733$01.00 © 1983 IEEE

Authorized licensed use limited to: to the Claremont Colleges!. Downloaded on June 11, 2009 at 17:52 from IEEE Xplore. Restrictions apply.